Control Hazard Pitfalls

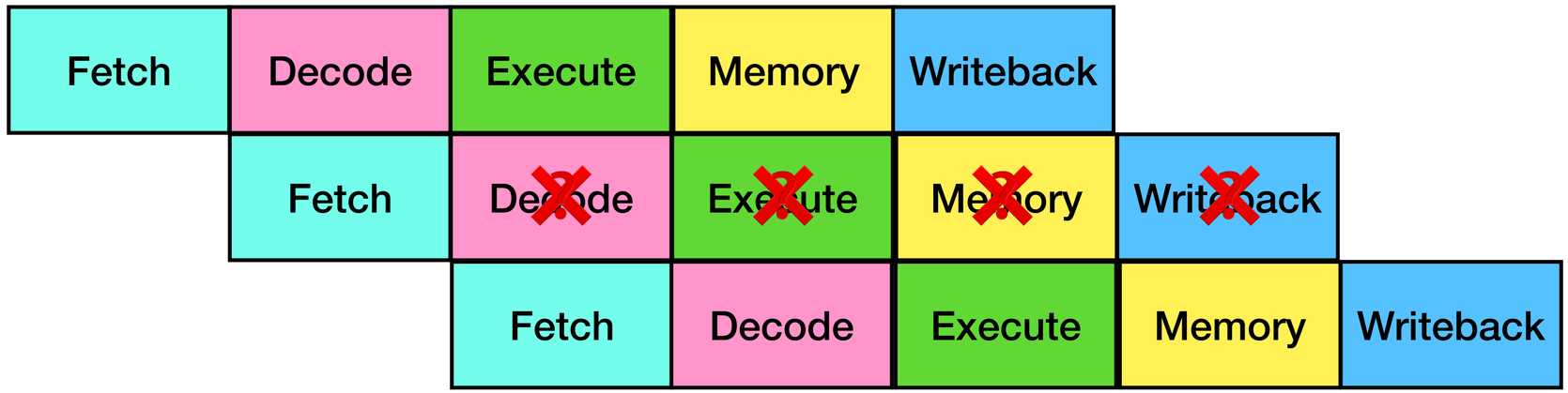
Colloquium today at Mudd;
Check-In 5 in class

Outline

- Revisiting and recontextualizing the control hazard setup
- How bad are control hazards?
- A strategy mitigate control hazards

From Wednesday: Control Hazards in the Data Path

```
1 bne r1, r2, 1
2 ldi r2, 30
3 end
```



When is the work done by the pipeline wasted work?

Mitigating Control Hazards

- In our data path, we have designed a system in which we can update the program counter after the decode stage for unconditional jumps and in the execute stage for conditional jumps
- Our hazard checking unit needs to flush the pipeline of any incorrect instructions that are currently residing in the pipeline
- For conditional jumps, it may be the case that the work in the pipeline is in fact the correct work to be doing ➡️ this happens when we *do not take* the branch (e.g., the program counter was correctly updated by the fetch stage)

The Impact of Control Hazards

```
1  ldi r1, 0
2  ldi r2, 1
3  ldi r3, 4
4  ldi r4, 7
5  ldi r5, 2
6  bge r1, r3, 3
7  add r1, r1, r2
8  mul r4, r4, r5
9  jmp -4
10 end
```

1 start 0, end 4				
2 start 1, end 5				
3 start 2, end 6				
4 start 3, end 7				
5 start 4, end 8				
6 start 5, end 9	6 start 10, end 14	6 start 15, end 19	6 start 20, end 24	6 start 25, end 29
7 start 6, end 10	7 start 11, end 15	7 start 16, end 20	7 start 21, end 25	
8 start 7, end 11	8 start 12, end 16	8 start 17, end 21	8 start 22, end 26	
9 start 8, end 12	9 start 13, end 17	9 start 18, end 22	9 start 23, end 27	
				10 start 29, end 33

Unconditional Jumps ➡️ stall until Decode completes (+2 cycles)

(Taken) Conditional Jumps ➡️ stall until Execute completes (+3 cycles)

Optimizing Around Control Hazards

```
int x = 7;
for (int i = 0; i < 4; i++) {
    x *= 2;
}
```

```
1  ldi r1, 0
2  ldi r2, 1
3  ldi r3, 4
4  ldi r4, 7
5  ldi r5, 2
6  bge r1, r3, 3
7  add r1, r1, r2
8  mul r4, r4, r5
9  jmp -4
10 end
```

```
int x = 7;
int i = 0;
x *= 2;
i++;
x *= 2;
i++;
x *= 2;
i++;
```

```
1  ldi r1, 0
2  ldi r2, 1
3  ldi r3, 4
4  ldi r4, 7
5  ldi r5, 2
6  mul r4, r4, r5
7  add r1, r1, r2
8  mul r4, r4, r5
9  add r1, r1, r2
10 mul r4, r4, r5
   add r1, r1, r2
   mul r4, r4, r5
   add r1, r1, r2
   end
```

Unrolled loop finishes at clock cycle 18 compared to finishing at clock cycle 39!

Some Observations…

- In our data path thus far, we always consider the *not taken* side of a conditional branch to be the default behavior ➡️ this is the straightforward approach to update the program counter for most instructions
- What if we could change our default behavior to set the default behavior to be *taken…* which common program behaviors exhibit this property?
- Our goal: ensure that the next instruction loaded into the pipeline is the right instruction to execute ➡️ no stalls due to control hazards!

```
FILE *file = fopen(path, “r”);
if (file == NULL) {
    printf(“Could not open file: %s”, path);
    return -1;
}

// file manipulation
```

Takeaways

- Unlike with data hazards, there is no easy fix to a control hazard… strategies like forwarding do not work as the correct resolution of the branch does not exist in the pipeline until it is resolved
- Mitigating control hazards means changing the structure of the program (e.g., through loop unrolling)
- If we can ensure that the correct next instruction to execute is being fetched/decoded by our pipeline, then we won't have to pay for costly control hazards… this will require (another) re-imagination of the data path!